LAB REPORT-2

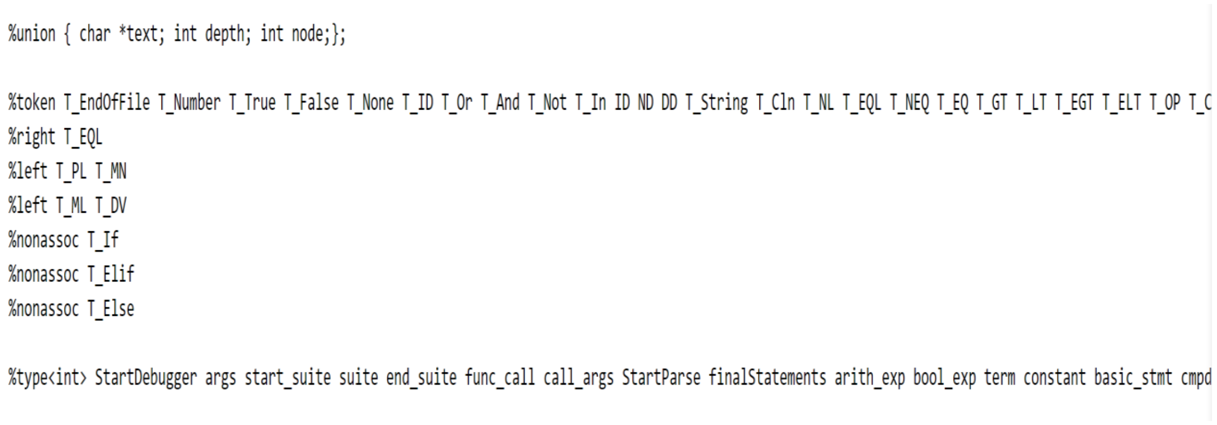
Subtopic: Parser

A. Create a parser for your programming language

B. Show that this parser correctly parses the input token generated by your lexical analyser for any program written in your programming language as well as identifies errors.

C. Write a simple program in your language with all kinds of tokens and keywords and show that your compiler is correctly detecting the tokens and errors. Parse the program using your parser. Print step by step parsing process and draw the parse tree.

A.Create a parser for your programming language.

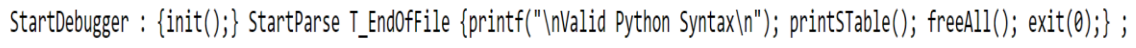


```
%union { char *text; int depth; int node;};

%token T_EndOfFile T_Number T_True T_False T_None T_ID T_Or T_And T_Not T_In ID ND DD T_String T_Cln T_NL T_EQL T_NEQ T_EQ T_GT T_LT T_EGT T_ELT T_OP T_C
%right T_EQL
%left T_PL T_MN
%left T_ML T_DV
%nonassoc T_If
%nonassoc T_Elif
%nonassoc T_Else

%type<int> StartDebugger args start_suite suite end_suite func_call call_args StartParse finalStatements arith_exp bool_exp term constant basic_stmt cmpd
```

This section defines the tokens recognized by the lexer (**%token**) and the types of the parser's semantic values (**%type**). **%union** defines the possible types of semantic values. **%right**, **%left**, and **%nonassoc** define the associativity and precedence of operators.

Yacc Production Rules and Grammar

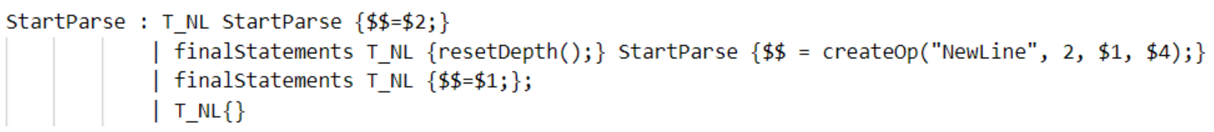
StartDebugger



```
StartDebugger : {init();} StartParse T_EndOfFile {printf("\nValid Python Syntax\n"); printSTable(); freeAll(); exit(0);} ;
```

- This rule initializes the parser by calling the **init()** function.
- It then starts parsing the input with the **StartParse** rule.
- When the end of the file is reached (**T_EndOfFile**), it prints a message indicating valid Python syntax, prints the symbol table, frees memory, and exits.

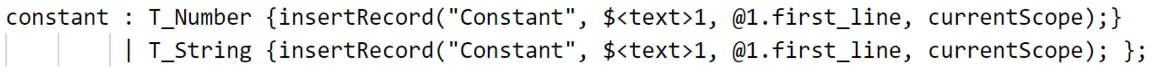
StartParse



```
StartParse : T_NL StartParse {$$=$2;}
           | finalStatements T_NL {resetDepth();} StartParse {$$ = createOp("NewLine", 2, $1, $4);}
           | finalStatements T_NL {$$=$1;};
           | T_NL{}
```

- This rule defines the starting point of parsing.
- It handles newlines (**T_NL**) and calls **resetDepth()** if needed to reset the parser depth.

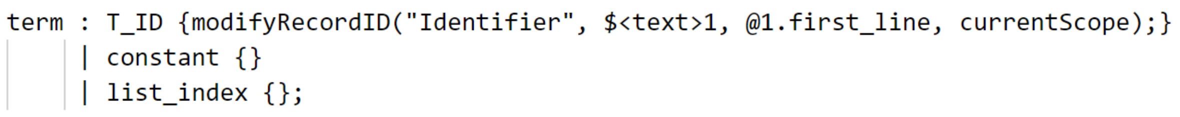
constant



```
constant : T_Number {insertRecord("Constant", $<text>1, @1.first_line, currentScope);}
         | T_String {insertRecord("Constant", $<text>1, @1.first_line, currentScope); };
```

- This rule defines how constants are parsed.
- It recognizes tokens representing numbers (**T_Number**) and strings (**T_String**).
- For each constant, it inserts a record into the symbol table with the appropriate type ("Constant"), the value of the constant, and the line number.

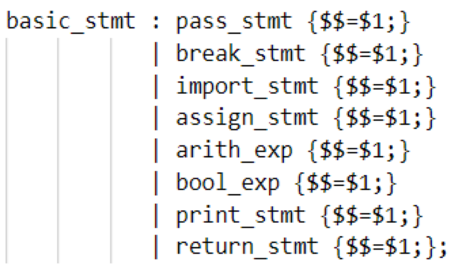
term



```
term : T_ID {modifyRecordID("Identifier", $<text>1, @1.first_line, currentScope);}
     | constant {}
     | list_index {};
```

- Terms represent individual elements in expressions.
- It can be an identifier (**T_ID**), a constant, or a list index.
- Depending on the type of the term, it modifies the record ID for identifiers, inserts constant records, or performs list index checks.

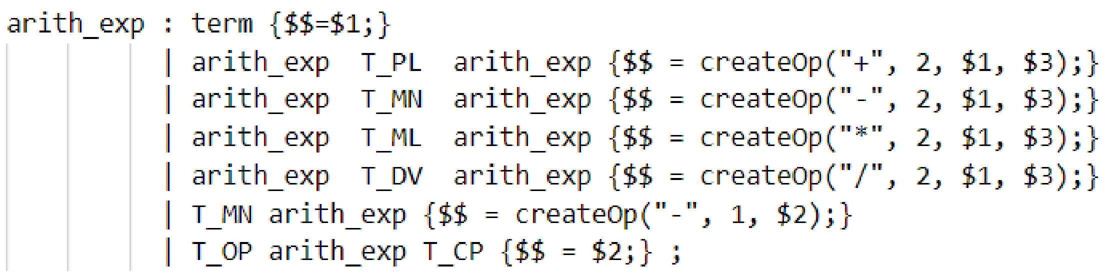
basic_stmt



```
basic_stmt : pass_stmt {$$=$1;}
           | break_stmt {$$=$1;}
           | import_stmt {$$=$1;}
           | assign_stmt {$$=$1;}
           | arith_exp {$$=$1;}
           | bool_exp {$$=$1;}
           | print_stmt {$$=$1;}
           | return_stmt {$$=$1;};
```

This rule defines various basic statements in Python, such as **pass_stmt**, **break_stmt**, **import_stmt**, **assign_stmt**, **arith_exp**, **bool_exp**, **print_stmt**, and **return_stmt**.

arith_exp



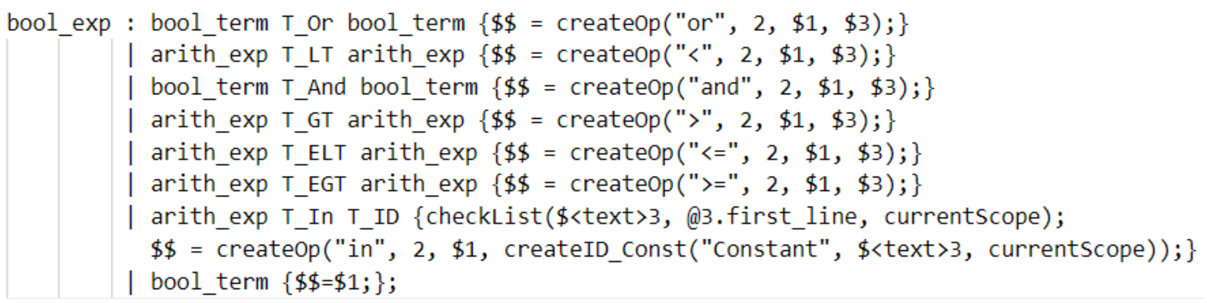
```
arith_exp : term {$$=$1;}
          | arith_exp  T_PL  arith_exp {$$ = createOp("+", 2, $1, $3);}
          | arith_exp  T_MN  arith_exp {$$ = createOp("-", 2, $1, $3);}
          | arith_exp  T_ML  arith_exp {$$ = createOp("*", 2, $1, $3);}
          | arith_exp  T_DV  arith_exp {$$ = createOp("/", 2, $1, $3);}
          | T_MN arith_exp {$$ = createOp("-", 1, $2);}
          | T_OP arith_exp T_CP {$$ = $2;} ;
```

- This rule defines arithmetic expressions, including addition, subtraction, multiplication, and division.

- It recursively defines how arithmetic expressions can be composed of other arithmetic expressions and terms.
- Parentheses can also be used to group expressions.

Each rule specifies how higher-level constructs are formed from lower-level constructs, following the syntax rules of the Python language. The associated actions with each rule may involve modifying the symbol table, performing semantic checks, or executing other operations necessary for parsing and interpreting Python code.

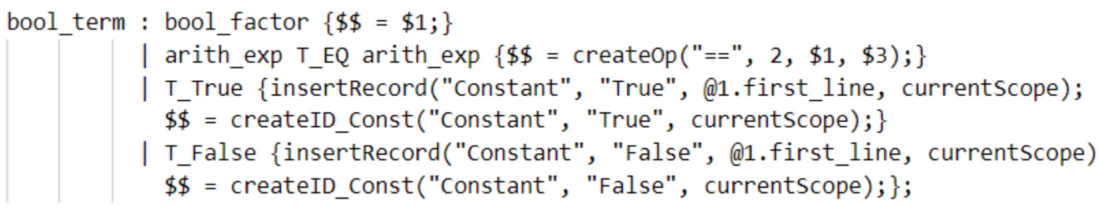
bool_exp:



```
bool_exp : bool_term T_Or bool_term {$$ = createOp("or", 2, $1, $3);}
         | arith_exp T_LT arith_exp {$$ = createOp("<", 2, $1, $3);}
         | bool_term T_And bool_term {$$ = createOp("and", 2, $1, $3);}
         | arith_exp T_GT arith_exp {$$ = createOp(">", 2, $1, $3);}
         | arith_exp T_ELT arith_exp {$$ = createOp("<=", 2, $1, $3);}
         | arith_exp T_EGT arith_exp {$$ = createOp(">=", 2, $1, $3);}
         | arith_exp T_In T_ID {checkList($<text>3, @3.first_line, currentScope);
           $$ = createOp("in", 2, $1, createID_Const("Constant", $<text>3, currentScope));}
         | bool_term {$$=$1;};
```

- This rule defines boolean expressions, which can be constructed using boolean terms (**bool_term**) and logical operators (**T_Or**, **T_And**).
- It also allows comparisons (**T_LT**, **T_GT**, **T_ELT**, **T_EGT**) between arithmetic expressions (**arith_exp**).
- Additionally, it supports membership checks (**T_In**) for elements in a collection (**T_ID**).
- The rule also allows for simple boolean terms without any logical or comparison operations.

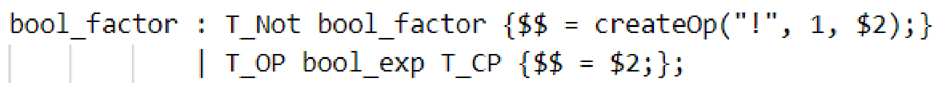
bool_term:



```
bool_term : bool_factor {$$ = $1;}
          | arith_exp T_EQ arith_exp {$$ = createOp("==", 2, $1, $3);}
          | T_True {insertRecord("Constant", "True", @1.first_line, currentScope);
            $$ = createID_Const("Constant", "True", currentScope);}
          | T_False {insertRecord("Constant", "False", @1.first_line, currentScope);
            $$ = createID_Const("Constant", "False", currentScope);};
```

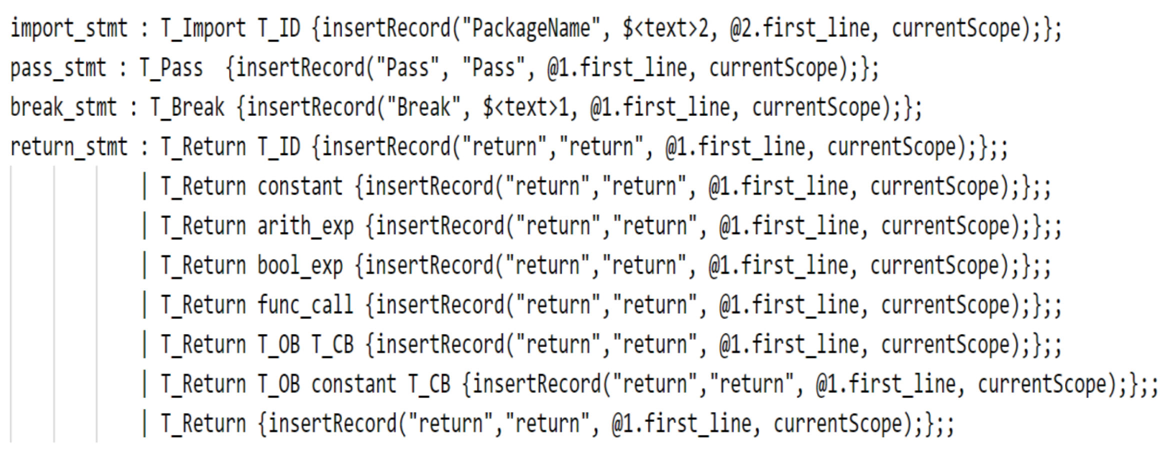
- Defines boolean terms, which can be boolean factors (**bool_factor**) or comparisons between arithmetic expressions (**arith_exp**) using equality (**T_EQ**).
- It also recognizes the boolean constants **True** and **False**, inserting corresponding records into the symbol table.

bool_factor:



```
bool_factor : T_Not bool_factor {$$ = createOp("!", 1, $2);}
            | T_OP bool_exp T_CP {$$ = $2;};
```

Represents boolean factors, which can be negated using the **T_Not** operator or enclosed within parentheses using **T_OP** and **T_CP**.



```
import_stmt : T_Import T_ID {insertRecord("PackageName", $<text>2, @2.first_line, currentScope);};
pass_stmt : T_Pass  {insertRecord("Pass", "Pass", @1.first_line, currentScope);};
break_stmt : T_Break {insertRecord("Break", $<text>1, @1.first_line, currentScope);};
return_stmt : T_Return T_ID {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return constant {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return arith_exp {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return bool_exp {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return func_call {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return T_OB T_CB {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return T_OB constant T_CB {insertRecord("return","return", @1.first_line, currentScope);};;
            | T_Return {insertRecord("return","return", @1.first_line, currentScope);};;
```

import_stmt:

- Handles import statements in Python.
- Recognizes the **T_Import** token followed by an identifier (**T_ID**), representing the package name being imported.
- Inserts a record into the symbol table with the package name.

pass_stmt:

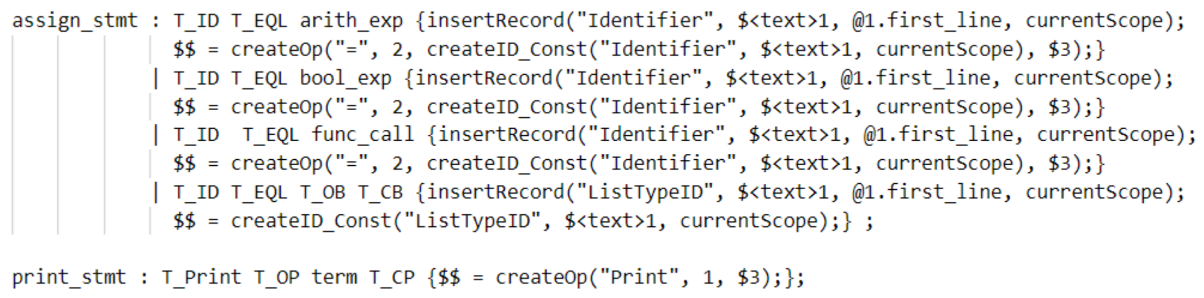
- Represents the **pass** statement, indicating no operation to be performed.
- Inserts a record with the value "Pass" into the symbol table.

break_stmt:

- Represents the **break** statement, used to exit a loop prematurely.
- Inserts a record with the value "Break" into the symbol table.

return_stmt:

- Handles **return** statements in functions.
- It can return an identifier (**T_ID**), a constant, an arithmetic expression, a boolean expression, a function call, an empty list, or a list with constants or expressions.
- Inserts a record with the value "return" into the symbol table.



```
assign_stmt : T_ID T_EQL arith_exp {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);
              $$ = createOp("=", 2, createID_Const("Identifier", $<text>1, currentScope), $3);}
            | T_ID T_EQL bool_exp {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);
              $$ = createOp("=", 2, createID_Const("Identifier", $<text>1, currentScope), $3);}
            | T_ID  T_EQL func_call {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);
              $$ = createOp("=", 2, createID_Const("Identifier", $<text>1, currentScope), $3);}
            | T_ID T_EQL T_OB T_CB {insertRecord("ListTypeID", $<text>1, @1.first_line, currentScope);
              $$ = createID_Const("ListTypeID", $<text>1, currentScope);} ;

print_stmt : T_Print T_OP term T_CP {$$ = createOp("Print", 1, $3);};
```

assign_stmt:

- Handles assignment statements where a value is assigned to a variable.
- It can assign arithmetic expressions (**arith_exp**), boolean expressions (**bool_exp**), function calls, or lists to identifiers (**T_ID**).
- Inserts records into the symbol table accordingly.

print_stmt:

- Represents **print** statements in Python, used to output text or variables to the console.
- Recognizes the **print** keyword followed by a list of terms enclosed in parentheses.
- No symbol table insertion is performed here; it's mainly for handling the printing action.

These yacc rules define the syntax and semantics for various constructs in the Python language, including expressions, statements, and control flow mechanisms. Each rule specifies how different elements are composed and processed during the parsing stage. Additionally, actions associated with rules perform tasks such as symbol table manipulation and semantic checks.

if:

```
if_stmt : T_If bool_exp T_Cln start_suite {$$ = createOp("If", 2, $2, $4);}
        | T_If bool_exp T_Cln start_suite elif_stmts {$$ = createOp("If", 3, $2, $4, $5);};
```

- Handles the parsing of **if** statements in Python.
- It starts with the **T_If** token followed by a boolean expression (**bool_exp**), a colon (**T_Cln**), and a suite of statements (**start_suite**).
- It can also have **elif** branches (**elif_stmts**) and an optional **else** branch (**else_stmt**).

elif and **else**:

```
elif_stmts : else_stmt {$$= $1;}
           | T_Elif bool_exp T_Cln start_suite elif_stmts {$$= createOp("Elif", 3, $2, $4, $5);};

else_stmt : T_Else T_Cln start_suite {$$ = createOp("Else", 1, $3);};
```

- **elif_stmts** represent the **elif** branches of an **if** statement.
- It starts with the **T_Elif** token followed by a boolean expression, a colon, and a suite of statements.
- **else_stmt** represents the **else** branch of an **if** statement.
- It starts with the **T_Else** token followed by a colon and a suite of statements.

while:

```
while_stmt : T_While bool_exp T_Cln start_suite {$$ = createOp("While", 2, $2, $4);};
```

- Handles the parsing of **while** loops in Python.
- It starts with the **T_While** token followed by a boolean expression, a colon, and a suite of statements (**start_suite**).

start_suite and **suite**:

```
start_suite : basic_stmt {}
            | T_NL ID {} finalStatements suite {};
            | T_NL {};

suite : T_NL ND finalStatements suite {}
      | T_NL end_suite {};
```

- **start_suite** is the beginning of a suite of statements.

- It can consist of basic statements, a newline followed by an identifier (**T_NL ID**), and then a set of final statements (**finalStatements**) followed by a suite.
- **suite** represents a suite of statements.
- It starts with a newline (**T_NL**) followed by the indentation depth (**ND**), a set of final statements, and another suite or an **end_suite**.

end_suite:

```
end_suite : DD {} finalStatements {}
          | DD {} {}
          | { resetDepth();};
```

- Represents the end of a suite of statements.
- It can consist of a decrease in indentation depth (**DD**), followed by a set of final statements or nothing, and optionally, a reset of the indentation depth (**resetDepth()**).

List:

```
list : T_OB elements T_CB | T_OB T_CB
elements : element | element T_Comma elements
element : bool_term | term | arith_exp | list
```

- This rule defines the structure of lists.
- It can be either an empty list (**T_OB T_CB**) or a list with elements enclosed in square brackets.
- The **elements** rule defines how elements are structured within the list.

list_index

```
list_index : T_ID T_OB constant T_CB {checkList($<text>1, @1.first_line, currentScope); };
```

- This rule handles list indexing, where an element in a list is accessed by its index.
- It checks the validity of the list index and updates the symbol table accordingly.

finalStatements:

```
finalStatements : basic_stmt {$$ = $1;}
                | cmpd_stmt {$$ = $1;}
                | func_def {$$ = $1;}
                | func_call {$$ = $1;}
                | decorator_def {$$ = $1;}
                | error T_NL {yyerrok; yyclearin; $$=createOp("SyntaxError", 0);};
```

- This rule represents the final set of statements or constructs that can conclude a block of code.
- It can consist of basic statements (**basic_stmt**), compound statements (**cmpd_stmt**), function definitions (**func_def**), decorator definitions (**decorator_def**), function calls (**func_call**), or error handling statements.
- If an error occurs during parsing, the **error** token followed by a newline (**T_NL**) triggers error handling actions (**yyerrok; yyclearin;**).

args and **args_list**:

```
args : T_ID {addToList($<text>1, 1);} args_list {$$ = createOp(argsList, 0); clearArgsList();}
     | {$$ = createOp("Void", 0);};

args_list : T_Comma T_ID {addToList($<text>2, 0);} args_list | ;
```

- These rules handle function arguments and their lists.
- **args** represent function arguments and can consist of an identifier followed by an **args_list**.
- **args_list** represents the list of arguments and can consist of a comma-separated list of identifiers.

call_list:

```
call_list : T_Comma term {addToList($<text>1, 0);} call_list | ;
```

- Handles the list of arguments in a function call.
- It consists of a comma-separated list of terms.

call_args:

```
call_args : T_ID {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0); clearArgsList();}
          | T_Number {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0); clearArgsList();}
          | T_String {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0); clearArgsList();}
          | {$$ = createOp("Void", 0);};
```

- This rule handles the arguments passed to a function during a function call.
- It can match one of the following patterns:
  - If the token is an identifier (**T_ID**), it adds the identifier to the argument list and clears the argument list with **clearArgsList()**.
  - If the token is a number (**T_Number**), it adds the number to the argument list and clears the argument list.
  - If the token is a string (**T_String**), it adds the string to the argument list and clears the argument list.
  - If there are no arguments, it matches an empty rule **{}**.

```
func_def : T_Def T_ID {insertRecord("Func_Name", $<text>2, @2.first_line, currentScope);} T_OP args T_CP T_Cln start_suite
         {$$ = createOp("Func_Name", 3, createID_Const("Func_Name", $<text>2, currentScope), $5, $8);};

decorator_def : T_Decorator T_ID {insertRecord("Decorator_Name", $<text>2, @2.first_line, currentScope);insertRecord("Decorator", "Decorator", @2.first_
```

func_def:

- This rule represents the definition of a function.
- It starts with the **T_Def** token followed by the function name (**T_ID**).
- It then inserts records into the symbol table for the function name and its definition.
- After that, it expects an opening parenthesis **T_OP**, followed by function arguments (**args**), a closing parenthesis **T_CP**, a colon **T_Cln**, and a suite of statements (**start_suite**).
- Before parsing the suite of statements, it updates the current scope, initializes a new symbol table, and then updates the scope again after parsing the suite.

decorator_def:

- This rule handles the definition of a decorator.
- It begins with the **T_Decorator** token followed by the decorator name (**T_ID**).
- Then, it inserts records for the decorator name and its definition into the symbol table.
- After that, it expects a newline (**T_NL**) followed by a function definition (**func_def**).

func_call:

```
func_call : T_ID T_OP call_args T_CP {$$ = createOp("Func_Call", 2, createID_Const("Func_Name", $<text>1, currentScope), $3);};
```

- This rule defines a function call.
- It starts with the function name (**T_ID**), followed by an opening parenthesis **T_OP**, the function arguments (**call_args**), and finally a closing parenthesis **T_CP**.

```
void yyerror(const char *msg)
{
    printf("\nSyntax Error at Line %d, Column : %d\n",  yylineno, yylloc.last_column);
    exit(0);
}

int main()
{
    //printf("Enter the Expression\n");
    yyparse();
    return 0;
}
```

This section defines the grammar rules using yacc syntax. Each rule specifies how higher-level constructs are formed from lower-level constructs. The **%%** separates the grammar rules from the C code. The **yyerror** function handles syntax errors encountered during parsing. The **main** function initiates the parsing process using **yyparse**.

Indent Depth Calculation

```
static int indent_depth(const char *K)
{
    int len = strlen(K), i, tab_count=1;
    for(i=0; i< len ; i++)
    {
        if(K[i]=='\t')
        {
            tab_count++;
        }
        else
        {
            printf("Nope");
            break;
        }
    }
    return tab_count;
}
int depth = 1;
```

This function calculates the depth of indentation based on the number of tabs (**\t**) encountered. It returns the count of tabs plus one (for one-based indexing). It is used for indentation-related tokenization.

Indentation Handling Rules

```
[\t]*       {
            depth = indent_depth(yytext);
            printf("Depth : %d ->",  depth);

            if(depth < top())
            {
                while (depth < top()) pop();
                yylval.depth = depth;
                show();
                debug("DD");
                return DD;
            }

            if(depth == top())
            {
                debug("ND");
                yylval.depth = depth;
                show();
                return ND;
            }
            if(depth > top())
            {
                push(depth);
                debug("ID");
                yylval.depth = depth;
                show();
                return ID;
            }

        }
```

This rule handles indentation in our programming language. It calculates the depth of indentation based on the number of tabs (**\t**) encountered. If the calculated depth is less than the top of the stack, it pops elements until the depth becomes equal to or less than the top. It returns different tokens (**DD**, **ND**, **ID**) based on indentation depth.

Symbol Table Helper Functions and Utility Definitions

```
  record* findRecord(const char *name, const char *type, int scope);
int *createID_Const(char *value, char *type, int scope);
void updateCScope(int scope);
void resetDepth();
  int scopeBasedTableSearch(int scope);
  void initNewTable();
  void init();
  int searchRecordInScope(const char* type, const char *name, int index);
  void insertRecord(const char* type, const char *name, int lineNo, int scope);
  void checkList(const char *name, int lineNo, int scope);
  void printSTable();
  void freeAll();
  void addToList(char *newVal, int flag);
  void clearArgsList();
```

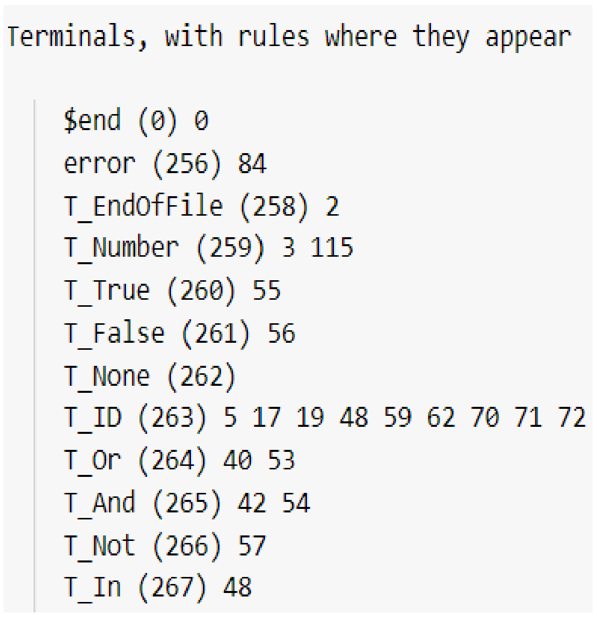
1. **addToList**: Appends a new value to a list. It takes a string **newVal** and a flag indicating whether to append the value or replace the existing list.
2. **clearArgsList**: Clears the contents of the argument list.
3. **updateCScope**: Updates the current scope based on the input parameter **scope**. If **scope** is -1, it resets the scope to 1; otherwise, it increments the current scope by **scope_Count**.
4. **resetDepth**: Resets the depth of a data structure, ensuring that it's empty.
5. **scopeBasedTableSearch**: Searches for a symbol table based on the scope provided as input. Returns the index of the symbol table if found, otherwise -1.
6. **initNewTable**: Initializes a new symbol table. It increments the symbol table index (**sIndex**), sets various attributes like scope, parent, and allocates memory for elements.
7. **findRecord**: Searches for a record with the given name, type, and scope in the symbol table.
8. **createID_Const**: Creates a new identifier constant with the given value, type, and scope.
9. **searchRecordInScope**: Searches for a record within a specific scope in the symbol table.
10. **insertRecord**: Inserts a new record with the given type, name, line number, and scope into the symbol table.
11. **checkList**: Checks if a given name is present in a list within the specified scope. It's used to validate whether an identifier is indexable.

B. Show that this parser correctly parses the input token generated by your lexical analyser for any program written in your programming language as well as identifies errors.

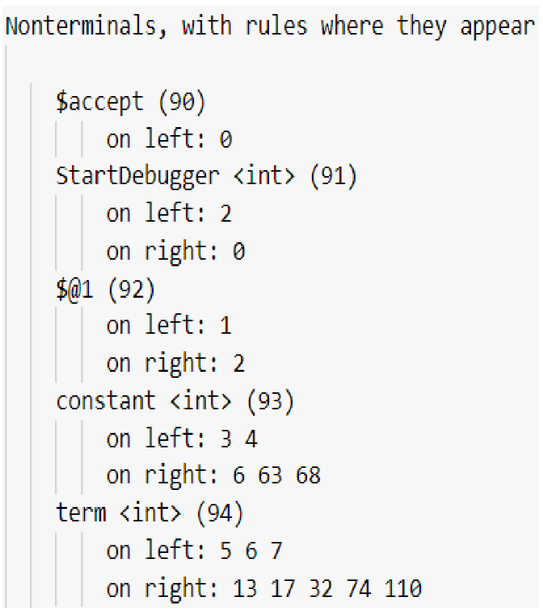
A Yacc-generated parser is a parser automatically generated based on a formal grammar specification provided by the user. Utilizing bottom-up parsing techniques like LALR (Look-Ahead Left-to-Right, Rightmost Derivation), these parsers efficiently construct parse trees by analyzing input tokens according to the grammar rules.

Parsing tables, generated from the grammar specification, guide the parser in determining appropriate actions, such as shift and reduce operations, during the parsing process. These parsers can detect syntax errors as well.

When using Yacc, the command yacc generates three primary files: y.tab.c, y.tab.h, and y.output. y.tab.c contains the C source code for the parser, y.tab.h declares data structures and function prototypes used by the parser, while y.output provides information about the grammar and parser states.



```
Terminals, with rules where they appear

    $end (0) 0
    error (256) 84
    T_EndOfFile (258) 2
    T_Number (259) 3 115
    T_True (260) 55
    T_False (261) 56
    T_None (262)
    T_ID (263) 5 17 19 48 59 62 70 71 72
    T_Or (264) 40 53
    T_And (265) 42 54
    T_Not (266) 57
    T_In (267) 48
```

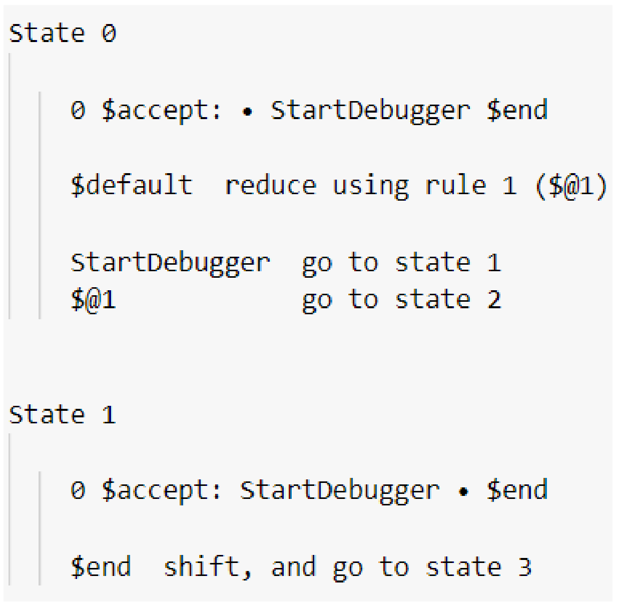


```
Nonterminals, with rules where they appear

    $accept (90)
        on left: 0
    StartDebugger <int> (91)
        on left: 2
        on right: 0
    $@1 (92)
        on left: 1
        on right: 2
    constant <int> (93)
        on left: 3 4
        on right: 6 63 68
    term <int> (94)
        on left: 5 6 7
        on right: 13 17 32 74 110
```

Defines the set of terminals produced by lexical analyzer.

Defines the set of non-terminals produced by lexical analyzer.

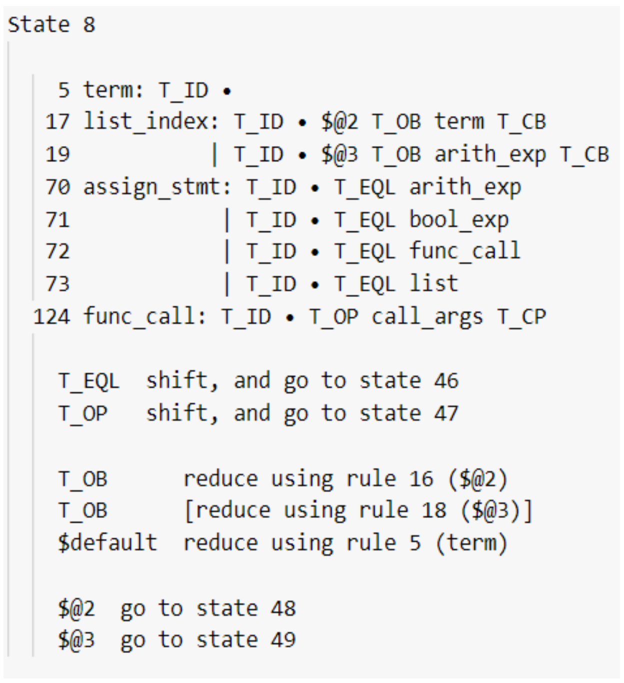
The above snippet is produced by y.output file which uses these symbols from symbol table along with state numbers where they are used.



```
State 0

    0 $accept: • StartDebugger $end

    $default  reduce using rule 1 ($@1)

    StartDebugger  go to state 1
    $@1            go to state 2


State 1

    0 $accept: StartDebugger • $end

    $end  shift, and go to state 3
```

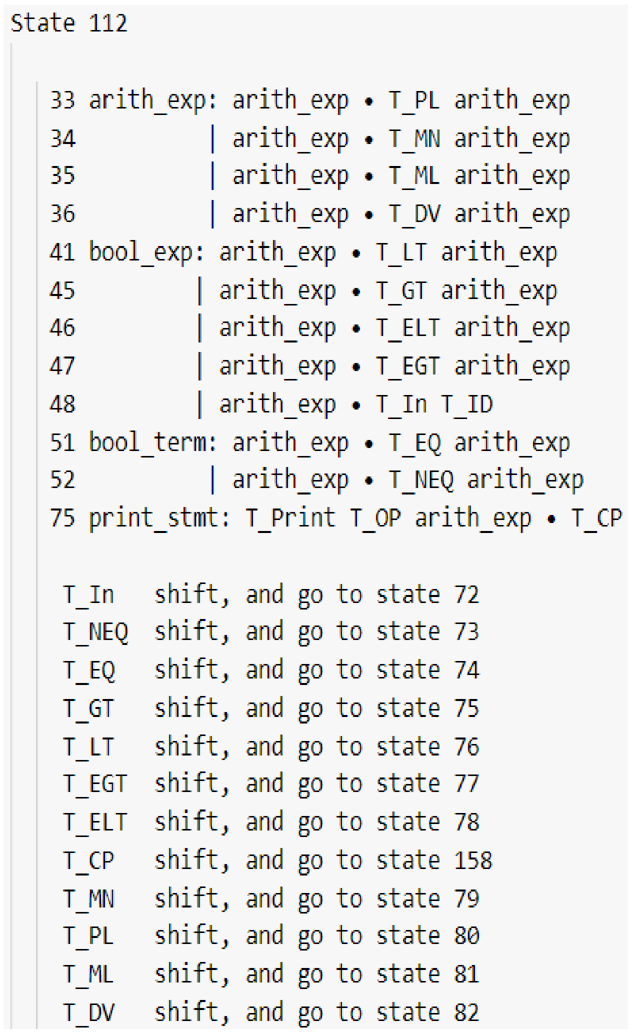
The y.output file generated by Yacc typically contains detailed information about the LALR(1) parsing automaton used during the parsing process. This automaton is of the LALR(1) type, meaning it performs bottom-up parsing while considering one token of lookahead. The file outlines the states of the LALR(1) automaton, transitions between states based on input tokens, lookahead symbols triggering transitions, and any conflicts encountered, such as shift-reduce or reduce-reduce conflicts.

The dot(.) at start of production rule indicates the parsing of current symbol in grammar and it mentions the shift/reduce conditions alongside the production rules.



```
State 8

  5 term: T_ID •
 17 list_index: T_ID • $@2 T_OB term T_CB
 19           | T_ID • $@3 T_OB arith_exp T_CB
 70 assign_stmt: T_ID • T_EQL arith_exp
 71            | T_ID • T_EQL bool_exp
 72            | T_ID • T_EQL func_call
 73            | T_ID • T_EQL list
124 func_call: T_ID • T_OP call_args T_CP

  T_EQL  shift, and go to state 46
  T_OP   shift, and go to state 47

  T_OB      reduce using rule 16 ($@2)
  T_OB      [reduce using rule 18 ($@3)]
  $default  reduce using rule 5 (term)

  $@2  go to state 48
  $@3  go to state 49
```

The excerpt from the y.output file describes State 8 of the LR(1) parsing automaton, outlining potential parsing configurations and productions. Each item signifies a point in a production where parsing has progressed. Actions like "shift" and "reduce" are defined for specific input tokens (e.g., T_EQL, T_OP), indicating transitions or reductions based on grammar rules. References to other states (e.g., 46, 47, 48, 49) show possible transitions based on lookahead symbols.



```
State 112

 33 arith_exp: arith_exp • T_PL arith_exp
 34          | arith_exp • T_MN arith_exp
 35          | arith_exp • T_ML arith_exp
 36          | arith_exp • T_DV arith_exp
 41 bool_exp: arith_exp • T_LT arith_exp
 45         | arith_exp • T_GT arith_exp
 46         | arith_exp • T_ELT arith_exp
 47         | arith_exp • T_EGT arith_exp
 48         | arith_exp • T_In T_ID
 51 bool_term: arith_exp • T_EQ arith_exp
 52          | arith_exp • T_NEQ arith_exp
 75 print_stmt: T_Print T_OP arith_exp • T_CP

  T_In   shift, and go to state 72
  T_NEQ  shift, and go to state 73
  T_EQ   shift, and go to state 74
  T_GT   shift, and go to state 75
  T_LT   shift, and go to state 76
  T_EGT  shift, and go to state 77
  T_ELT  shift, and go to state 78
  T_CP   shift, and go to state 158
  T_MN   shift, and go to state 79
  T_PL   shift, and go to state 80
  T_ML   shift, and go to state 81
  T_DV   shift, and go to state 82
```

In the given Yacc output excerpt, parsing state 112 illustrates the parser's progress in parsing arithmetic and boolean expressions, along with print statements. Each listed lookahead token triggers a shift action, directing the parser to the next state. For instance, encountering tokens like T_In or T_NEQ would shift the parser to states 72 or 73, respectively.

```
State 152

   87 if_stmt: T_If bool_exp T_Cln start_suite •
   88        | T_If bool_exp T_Cln start_suite • elif_stmts

    T_Else  shift, and go to state 171
    T_Elif  shift, and go to state 172

    T_Else    [reduce using rule 87 (if_stmt)]
    T_Elif    [reduce using rule 87 (if_stmt)]
    $default  reduce using rule 87 (if_stmt)

    elif_stmts  go to state 173
    else_stmt   go to state 174


State 153

   92 while_stmt: T_While bool_exp T_Cln start_suite •

    $default  reduce using rule 92 (while_stmt)
```

The excerpt from the y.output file depicts two states of the LR(1) parsing automaton. In State 152, the parser expects an if statement (if_stmt) followed by a boolean expression and a colon, with transitions possible upon encountering T_Else or T_Elif. It can also reduce the if statement rule if neither is present. In State 153, the parser anticipates a while statement (while_stmt) after encountering T_While, a boolean expression, and a colon, with a default reduction if no further tokens appear.

```
State 154

  121 func_def: T_Def T_ID $@12 T_OP • args T_CP T_Cln $@13 start_suite

    T_ID  shift, and go to state 175

    $default  reduce using rule 105 (args)

    args  go to state 176
```

In state 154, the parser is processing a function definition (func_def) with encountered tokens T_Def, T_ID, and T_OP. The current lookahead token is T_OP. At this state, the parser can shift T_ID to transition to state 175, reduce using rule 105 (args), or proceed to state 176 to parse additional arguments.

```
State 161

  123 decorator_def: T_Decorator T_ID $@14 T_NL • func_def

    T_Def  shift, and go to state 19

    func_def  go to state 177
```

State 161 is associated with the parsing of a decorator_def production, where the parser expects to encounter a T_Decorator, followed by an T_ID, and then a T_NL token.Following the dot, the parser expects to transition to the func_def non-terminal. The information provided indicates that upon encountering a T_Def token, the parser should shift and move to state 19, while for the func_def non-terminal, it should transition to state 177.

PARSING ERRORS

Sample Error Program

```
print 1+2
def f1():
  x=5
  while(x<10)
    if(xx==10):
    return x
  return x+2
print(f1()
```

```
---------------------------------Parsing Output-----------------------
1 Syntax Error at Line 1, Column : 6
T_NL
T_NL
T_NL
3 Syntax Error at Line 4, Column : 13
T_NL
4 T_If T_OP T_xx T_EQ Identifier 'xx' at line 5 Not Declared
T_NL
5 T_x Syntax Error at Line 6, Column : 4
T_NL
T_NL
Syntax Error at Line 8, Column : 10
T_EOF
```

The above code snippet contains parsing errors such as missing tokens, syntax rule mismatch and indentation errors. Our parser identifies these errors and generates syntax error stating line number and column number from yyerror function.

The following parser errors are identified in above code:

In line 1, print statement doesn't have parenthesis.
In line 4, while statement has missing colon.
In line 5, if statement has a condition where undeclared variable is used.
In line 6, return statement doesn't follow indentation rules.
In line 8, function call in print statement doesn't end with parenthesis.

C. Write a simple program in your language with all kinds of tokens and keywords

```
import math
x=0
y=10
i=0
#Comment1
print(x+y)
lis = [x,y]
```

This code contains import keyword, variable declarations, list, and comments and below is its parse tree.

```
StartParse
  |
  |--- T_Import
  |       |
  |       |--- T_ID
  |
  |--- assign_stmt
  |       |
  |       |--- T_ID (id='x')
  |       |--- T_EQL
  |       |--- T_NUMBER (Num=0)
  |
  |--- assign_stmt
  |       |
  |       |--- T_ID (id='y')
  |       |--- T_EQL
  |       |--- T_NUMBER (Num=10)
  |
  |--- assign_stmt
  |       |
  |       |--- T_ID (id='i')
  |       |--- T_EQL
  |       |--- T_NUMBER (Num=0)
  |
  |--- print_stmt
  |       |
  |       |--- T_Print
  |       |--- T_OP
  |       |--- term
  |               |
  |               |--- T_ID (id='x')
  |               |--- T_PL
  |               |--- T_ID (id='y')
  |       |--- T_CP
  |
  |--- assign_stmt
  |       |
  |       |--- T_ID (id='lis')
  |       |--- T_EQL
  |       |--- T_OB
  |       |--- elements
  |               |--- term[0] (id='x')
  |               |--- term[1] (id='y')
  |       |--- T_CB
```

```
def F1(A, B, C):
        a=1+x
        while(lis[a]==y):#Comment
                pass
                t=1
        z=10
        b=i
        if(z==b):
                c=10+b
        else:
                c=5+z
        return c
```

This code snippet includes function, while loop with list-indexing based condition, if-else declarations with various levels of indentation along with pass and return keywords.

```
  |--- func_def
  |       |
  |       |--- T_Def
  |       |--- T_ID (id='F1')
  |       |--- args
  |              |
  |              |--- args[0] (id='A')
  |              |--- args[1] (id='B')
  |              |--- args[2] (id='C')
  |       |--- T_Colon
  |
  |       |--- body
  |              |
  |              |--- assign_stmt
  |                     |
  |                     |--- T_ID (id='a')
  |                     |--- T_EQL
  |                     |--- arith_exp
  |                            |
  |                            |--- T_NUMBER (Num=1)
  |                            |--- T_PL
  |                            |--- T_ID (id='x')
  |              |
  |              |--- T_While
  |                     |
  |                     |--- bool_exp
  |                            |
  |                            |--- T_ID (id='lis')
  |                            |--- list_index
  |                                          |--- T_ID (id='a')
  |                            |--- T_EQ
  |                            |--- T_ID (id='y')
  |                     |--- body T_Pass
  |              |
  |              |--- assign_stmt
  |                     |
  |                     |--- T_ID (id='z')
  |                     |--- T_EQL
  |                     |--- T_NUMBER (Num=10)
  |              |
  |              |--- assign_stmt
```

```
|           |
|           |--- T_ID (id='b')
|           |--- T_EQL
|           |--- T_ID (id='i')
|       |
|       |--- if_stmt
|           |
|           |--- T_If
|               |
|               |--- bool_exp
|                   |
|                   |--- T_ID (id='z')
|                   |--- T_EQ
|                   |--- T_ID (id='b')
|               |--- T_Colon
|               |--- body
|                   |
|                   |--- assign_stmt
|                       |
|                       |--- T_ID (id='c')
|                       |--- T_EQL
|                       |--- arith_exp
|                           |
|                           |--- T_NUMBER (Num=10)
|                           |--- T_PL
|                           |--- T_ID (id='b')
|            |--- T_Else
|               |--- T_Colon
|               |--- body
|                   |
|                   |--- assign_stmt
|                       |
|                       |--- T_ID (id='c')
|                       |--- T_EQL
|                       |--- arith_exp
|                           |
|                           |--- T_NUMBER (Num=5)
|                           |--- T_PL
|                           |--- T_ID (id='z')
|       |--- T_Return
|       |   |
|       |   |--- T_ID (id='c')
|
```

```
@F1
def F2(A):
	x=y+1
	return
m1 = F1(10, 10, 10)
m2 = F2(0, 1, 2)
if(m1==m2):
	print(True)
else:
	print(False)
```

This code snippet involves decorator declaration and function calls.

```
|--- decorator_def
|       |
|       |--- T_Decorator
|       |--- T_ID (id='F1')
|
|--- func_def
|       |
|       |--- T_Def
|       |--- T_ID (id='F2')
|       |--- args
|              |
|              |--- args[0] (id='A')
|       |--- T_Colon
|
|       |--- body
|              |
|              |--- assign_stmt
|                         |
|                         |--- T_ID (id='x')
|                         |--- T_EQL
|                         |--- arith_exp
|                                    |
|                                    |--- T_ID(id='y')
|                                    |--- T_PL
|                                    |--- T_NUMBER (Num=1)
|              |--- T_Return
|
|--- assign_stmt
|       |
|       |--- T_ID (id='m1')
|       |--- T_EQL
|                 |
|                 |--- func (id='F1')
|                 |--- call_args (Num=10, Num=10, Num=10)
|       |
|--- assign_stmt
|       |
|       |--- T_ID (id='m2')
|       |--- T_EQL
|                 |
|                 |--- func (id='F2')
|                 |--- call_args (Num=0, Num=1, Num=2)
|       |
|--- T_If
|       |
|       |--- bool_exp
|                   |
|                   |--- T_ID (id='m1')
|                   |--- T_EQ
|                   |--- T_ID (id='m2')
|        |--- T_Colon
|        |--- body
|                   |
|                   |--- print_stmt
|                     |
|                     |--- T_Print
|                     |--- T_OP
|                     |--- term
|                           |
|                           |--- T_True
|                     |--- T_CP
|
|      |--- T_Else
|      |--- T_Colon
|      |--- body
|             |
|             |--- print_stmt
|                     |
|                     |--- T_Print
|                     |--- T_OP
|                     |--- term
|                           |
|                           |--- T_False
|                     |--- T_CP
```

Parser Output as Parsing Table:

State	T_ID	T_Def	'T_While	… T_EOF	StartParse	arith_exp	term	….
-------	------------	------------	------------	-------------	--------------	------------	------	….
0					1			
1				accept				
2	shift 8	shift 19	shift 17		26	28	24	
3	reduce R9							
4						28		
…								
204								
205		shift 19	shift 17			28	24	
206	shift 16							
207		shift 19	shift 17		35			
208	reduce R100							

The parsing table consists of 44 terminals, 50 non terminals and 160 states. The above parsing table (contains only few states for illustrative purposes, actual table is too large to fit in) obtained as parser output is used to efficiently analyse input according to the grammar rules of a language. It comprises rows representing parser states and columns representing input symbols. Each entry in the table specifies an action for the parser, such as shifting input onto a stack, reducing symbols based on grammar rules, or transitioning to new states. These tables play a critical role in enforcing syntactic correctness and detecting errors in input streams.

Parser Output as Symbol Table:

The symbol table within a parser stores details such as its name, type, scope, and declaration. This information is crucial for subsequent compiler phases, including semantic analysis and code generation, as it aids in resolving symbol references and ensuring program correctness.

Valid Python Syntax

```
---------------------------All Symbol Tables---------------------------
        Lexeme Token   Declaration  Scope
Scope : 1
1       math   PackageName 1        (1)
1       0      Constant    2        (1)
1       x      Identifier  2        (1)
1       y      Identifier  3        (1)
1       10     Constant    3        (1)
1       i      Identifier  4        (1)
1       lis    ListTypeID  7        (1)
1       F1     Func_Name   9        (1)
1       Def    Def         9        (1)
1       F1     DecoratorName23      (1)
1       DecoratorDecorator 23       (1)
1       F2     Func_Name   24       (1)
1       m1     Identifier  30       (1)
1       1      Constant    31       (1)
1       2      Constant    31       (1)
1       m2     Identifier  31       (1)
1       If-Elif If-Elif    32       (1)
1       True   Constant    33       (1)
1       Else   Else        34       (1)
1       False  Constant    35       (1)

Scope : 2
2       1      Constant    10       (2)
2       a      Identifier  10       (2)
2       While  While       11       (2)
2       Pass   Pass        12       (2)
2       t      Identifier  13       (2)
2       10     Constant    14       (2)
2       z      Identifier  14       (2)
2       b      Identifier  15       (2)
2       return return      16       (2)
2       c      Identifier  17       (2)
2       If-Elif If-Elif    18       (2)
2       5      Constant    19       (2)
2       Else   Else        20       (2)

Scope : 3
3       x      Identifier  25       (3)
3       1      Constant    25       (3)
3       return return      26       (3)
-----------------------------------------------------------------------
```

The scope defines the level of depth for indentation purposes.